Chapitre 3. Applications avancées des processeurs ARM-Cortex

3.1 Introduction générale

Les processeurs ARM Cortex, en particulier les familles Cortex-M et Cortex-A selon le domaine visé, occupent une place centrale dans les systèmes embarqués modernes grâce à leur bon compromis entre performance, consommation d'énergie, coût et richesse de l'écosystème logiciel . Dans les applications avancées, ils ne servent pas seulement à exécuter un programme séquentiel, mais deviennent le cœur de plateformes temps réel, de nœuds connectés pour l'Internet des objets et de périphériques capables de dialoguer via des interfaces normalisées comme l'USB . Ce chapitre présente trois axes majeurs d'utilisation avancée des processeurs ARM-Cortex : l'intégration d'un système d'exploitation temps réel avec FreeRTOS, le développement de solutions IoT, et le développement USB côté périphérique ou hôte. Ces trois thématiques sont complémentaires, car dans un système réel il est fréquent de combiner un ordonnancement temps réel, une connectivité réseau et des échanges filaires avec un ordinateur ou un autre équipement.

3.2 FreeRTOS

A. Définition et rôle

FreeRTOS est un noyau de système d'exploitation temps réel destiné aux dispositifs embarqués ; il est conçu pour être léger, rapide et portable sur de nombreuses architectures de microcontrôleurs. Son intérêt dans le contexte ARM Cortex-M est majeur, car il permet de structurer une application complexe en plusieurs tâches concurrentes, au lieu de recourir à une seule boucle infinie difficile à maintenir.

Dans une application bare-metal classique, toutes les fonctions sont souvent enchaînées manuellement dans la boucle principale, ce qui complique la gestion des priorités, des délais et des événements asynchrones. Avec FreeRTOS, le concepteur découpe le programme en tâches indépendantes, chacune disposant de sa propre pile et de sa priorité, tandis que le noyau se charge de l'ordonnancement.

B. Pourquoi FreeRTOS sur ARM Cortex-M

Les cœurs ARM Cortex-M sont particulièrement adaptés à FreeRTOS grâce à leurs mécanismes matériels d'interruptions et d'exceptions, qui facilitent les changements de contexte et la gestion d'un système temps réel préemptif. La documentation FreeRTOS dédiée aux cœurs Cortex-M met d'ailleurs l'accent sur l'importance du mécanisme de priorité d'interruptions, car une configuration incorrecte peut provoquer des dysfonctionnements subtils dans l'appel des services du noyau depuis les ISR.

Sur plusieurs familles de microcontrôleurs ARM, des ports FreeRTOS sont déjà disponibles, notamment pour des cœurs Cortex-M0, M0+, M4, M7 et M33. Cela réduit fortement l'effort d'intégration, car le développeur peut se concentrer sur la logique applicative, la configuration de l'horloge, des périphériques et des paramètres du noyau via le fichier FreeRTOSConfig.h.

C. Architecture de base de FreeRTOS

L'architecture fonctionnelle de FreeRTOS repose sur quelques briques essentielles : le planificateur, les tâches, les files de messages, les sémaphores, les mutex, les temporisations logicielles et éventuellement les timers logiciels. Le planificateur choisit à tout instant la tâche prête la plus prioritaire, ce qui permet de garantir une bonne réactivité pour les fonctions critiques comme l'acquisition de données, la commande moteur ou la surveillance d'alarmes.

Une tâche FreeRTOS correspond à une unité d'exécution indépendante. Elle est généralement écrite sous la forme d'une fonction infinie contenant un traitement périodique, événementiel ou mixte, avec des appels de blocage comme vTaskDelay() ou des attentes sur des objets de synchronisation. Ce modèle aide à séparer clairement les fonctions d'un système embarqué : interface utilisateur, communication série, lecture capteurs, traitement numérique et journalisation.

D. Ordonnancement et gestion des priorités

FreeRTOS peut fonctionner avec un ordonnancement préemptif ou coopératif selon la configuration du noyau . En mode préemptif, lorsqu'une tâche de priorité plus élevée devient

prête, elle prend la main automatiquement, ce qui est essentiel dans les applications temps réel où certains traitements doivent être exécutés avec une latence minimale.

Le découpage par priorités doit toutefois être réalisé avec méthode. Si trop de tâches sont affectées à des priorités élevées, le système peut devenir difficile à équilibrer et les tâches de fond risquent d'être affamées ; à l'inverse, des priorités trop basses pour une fonction critique peuvent compromettre les contraintes temporelles. Une bonne pratique consiste à réserver les plus fortes priorités aux traitements les plus sensibles au temps, et à laisser les fonctions de maintenance ou de communication non critiques à des niveaux inférieurs.

E. Communication inter-tâches

Dans un système embarqué avancé, les tâches ne travaillent jamais isolément. FreeRTOS propose plusieurs mécanismes de communication et de synchronisation, notamment les queues, les sémaphores et les mutex . Les queues sont très utiles pour transmettre des données entre une tâche productrice et une tâche consommatrice, par exemple entre une ISR d'acquisition et une tâche de traitement .

Les sémaphores binaires servent souvent à signaler un événement, tandis que les mutex sont adaptés à la protection de ressources partagées comme un bus I2C, une mémoire externe ou un terminal UART. L'usage correct de ces mécanismes évite les conflits d'accès, les corruptions de données et une partie des comportements non déterministes typiques des programmes concurrents .

F. Gestion du temps dans FreeRTOS

La notion de temps est fondamentale dans FreeRTOS. Le noyau s'appuie généralement sur une interruption d'horloge périodique, appelée tick, qui rythme les temporisations et certaines opérations d'ordonnancement . FreeRTOS propose aussi une option dite tick-less afin de réduire la consommation énergétique dans les applications basse consommation, ce qui est particulièrement intéressant pour les objets connectés alimentés par batterie .

Le développeur peut ainsi créer des tâches périodiques, temporiser une opération, surveiller un timeout de communication ou déclencher un traitement différé. Il doit cependant

choisir avec soin la fréquence du tick, car une valeur trop élevée augmente la charge système tandis qu'une valeur trop faible réduit la résolution temporelle .

G. Configuration pratique sur ARM Cortex

Dans un projet ARM Cortex-M, l'intégration de FreeRTOS passe souvent par l'ajout des fichiers source du noyau, la sélection du port matériel adapté au cœur utilisé et la personnalisation du fichier FreeRTOSConfig.h . Ce fichier centralise de nombreux paramètres : mode d'ordonnancement, fréquence de tick, taille du tas mémoire, nombre de priorités, activation des hooks et services API disponibles .

La configuration des interruptions est un point particulièrement sensible sur Cortex-M. La documentation FreeRTOS dédiée à cette architecture rappelle que la hiérarchie des priorités d'interruptions doit être cohérente avec les niveaux autorisés pour appeler les API depuis une routine d'interruption . Cela signifie qu'un système fonctionnel ne dépend pas seulement du code applicatif, mais aussi d'une compréhension fine de l'architecture NVIC et des priorités effectives .

H. Exemple pédagogique d'application

Considérons un système ARM Cortex-M destiné à la surveillance environnementale. Une première tâche peut acquérir périodiquement les mesures de température et d'humidité, une deuxième tâche peut préparer les données, une troisième peut transmettre les résultats par liaison série ou réseau, et une quatrième peut gérer les alarmes . Avec FreeRTOS, chaque fonction devient modulaire, testable et plus facile à maintenir qu'une architecture monolithique .

Dans ce scénario, les interruptions de capteurs ou de communication peuvent réveiller certaines tâches via des mécanismes adaptés, tandis que l'ordonnanceur garantit que les traitements critiques sont servis en priorité . Cette organisation illustre la vraie valeur de FreeRTOS : transformer un microcontrôleur en plateforme logicielle robuste, capable de gérer plusieurs activités concurrentes de manière déterministe .

I. Avantages et limites

Les principaux avantages de FreeRTOS sont sa faible empreinte mémoire, sa rapidité d'exécution, sa large diffusion dans le monde embarqué et son adéquation avec les

microcontrôleurs ARM Cortex . Il facilite la structuration logicielle, améliore la maintenabilité et favorise la réutilisation de modules dans des applications industrielles, domotiques, médicales ou IoT .

En revanche, l'usage d'un RTOS ne supprime pas automatiquement tous les problèmes de conception. Une mauvaise répartition des priorités, un dimensionnement insuffisant des piles, un usage abusif des sections critiques ou une synchronisation mal pensée peuvent produire des bugs complexes, parfois plus difficiles à diagnostiquer qu'en bare-metal .

3.3 Développement IoT

A. Notion d'IoT et place des ARM Cortex

L'Internet des objets, ou IoT, désigne l'ensemble des systèmes capables de capter des informations, de les traiter localement et de les transmettre à d'autres équipements ou à des services distants via un réseau . Les processeurs ARM Cortex occupent une position privilégiée dans ce domaine, car ils offrent un excellent compromis entre performances, consommation et intégration de périphériques de communication .

Dans un objet connecté moderne, le microcontrôleur ne se limite pas à lire des capteurs. Il doit aussi gérer la communication réseau, parfois le chiffrement, les mises à jour logicielles, la gestion d'énergie, et souvent une logique applicative temps réel . Cette convergence explique pourquoi les développements IoT s'appuient souvent sur un RTOS comme FreeRTOS pour organiser les différentes fonctions du système .

B. Chaîne fonctionnelle d'un objet connecté

Une architecture IoT basée sur ARM Cortex peut être décrite comme une chaîne de traitement en plusieurs niveaux : acquisition, traitement local, communication, supervision et éventuellement actionnement. Les données issues de capteurs sont d'abord acquises via des interfaces comme ADC, I2C, SPI ou UART, puis filtrées ou agrégées par le microcontrôleur .

Ensuite, les informations peuvent être envoyées vers une passerelle, un smartphone, un serveur local ou une plateforme cloud selon le protocole choisi. Le système reçoit aussi parfois des

commandes distantes, par exemple pour modifier une consigne, mettre à jour un paramètre ou déclencher un actionneur .

C. Protocoles et connectivité

Le développement IoT sur ARM Cortex repose sur des liaisons physiques et des protocoles réseau variés. Selon l'application, le nœud peut utiliser le Wi-Fi, le Bluetooth Low Energy, Ethernet, ou encore des communications bas débit longue portée via une passerelle dédiée . Le choix dépend de la portée, du débit, de la consommation énergétique, du coût et des exigences de sécurité du système .

Au niveau logiciel, l'application embarquée doit souvent gérer plusieurs couches : pilote matériel, pile de communication, protocole applicatif et logique métier. Cette superposition renforce l'intérêt d'une architecture multitâche, où une tâche s'occupe de l'acquisition, une autre de la connectivité, une autre des échanges applicatifs et une autre de la maintenance système .

3.4 FreeRTOS comme base logicielle de l'IoT

Sur plusieurs plateformes de développement orientées objet connecté, FreeRTOS est déjà intégré à l'environnement logiciel. Par exemple, la documentation ESP-IDF indique que les applications ESP-IDF sont fondées sur FreeRTOS comme composant central du système . Même si l'ESP32 n'est pas un cœur ARM, cet exemple montre le rôle structurant d'un RTOS dans la conception de produits IoT embarqués .

Dans le cas d'un microcontrôleur ARM Cortex-M connecté, FreeRTOS peut gérer les tâches périodiques, les files d'événements, les temporisations réseau et l'accès concurrent aux interfaces de communication. Cette approche simplifie le développement de nœuds intelligents capables de mesurer, décider et transmettre de manière fiable .

A. Gestion de l'énergie

L'un des enjeux majeurs de l'IoT est la consommation. Un objet connecté sur batterie doit passer une grande partie de son temps en veille, ne se réveiller qu'en cas d'événement ou à intervalles choisis, puis retourner rapidement dans un état basse consommation . FreeRTOS

propose une option tick-less destinée précisément à réduire l'activité inutile du processeur lorsque le système n'a aucune tâche prête à exécuter .

L'architecture ARM Cortex-M complète bien cette approche, car ces microcontrôleurs disposent généralement de modes sommeil, arrêt partiel ou veille profonde, associés à des réveils par interruption. Le développement IoT exige donc une co-optimisation entre logiciel temps réel, configuration des périphériques et stratégie énergétique .

B. Sécurité et fiabilité

Un système IoT ne doit pas seulement être connecté ; il doit aussi être fiable et résistant aux erreurs. La multiplication des échanges réseau augmente la surface d'attaque, ce qui impose une attention particulière à l'authentification, à l'intégrité des données, au chiffrement et à la gestion des mises à jour .

Sur le plan embarqué, la robustesse dépend également de la maîtrise du parallélisme, des temporisations et des reprises après erreur. Un ARM Cortex piloté par FreeRTOS permet d'isoler les responsabilités entre tâches, de surveiller les événements et de mieux organiser les mécanismes de récupération, mais cela reste conditionné par une bonne conception logicielle .

C. Cas d'usage typiques

Les applications IoT basées sur ARM Cortex couvrent un spectre très large : maison intelligente, agriculture de précision, maintenance industrielle, télémétrie énergétique, instrumentation médicale portable et suivi environnemental . Dans chacun de ces cas, le processeur local doit assurer simultanément la lecture des capteurs, le prétraitement, la communication et parfois une prise de décision locale en temps réel .

Par exemple, un nœud de surveillance agricole peut mesurer l'humidité du sol, transmettre les données à une passerelle, puis activer localement une électrovanne si un seuil critique est détecté. Une telle architecture combine précisément les trois thèmes de ce chapitre : RTOS, connectivité IoT et dialogue possible avec des interfaces de maintenance comme l'USB .

3.5 Développement USB

A. Rôle de l'USB dans les systèmes ARM Cortex

L'USB est une interface standardisée très utilisée dans les systèmes embarqués pour assurer la communication entre un microcontrôleur et un ordinateur, un smartphone, une mémoire de masse ou d'autres équipements compatibles. Dans les plateformes ARM Cortex disposant d'un contrôleur USB intégré, cette interface est souvent exploitée pour le débogage, l'échange de données, la mise à jour du firmware ou l'émulation de périphériques standards .

Le développement USB constitue une application avancée, car il ne s'agit pas simplement d'activer une broche de communication. Il faut respecter une pile protocolaire, des descripteurs, des classes de périphériques, un mécanisme d'énumération et une interaction précise entre la couche matérielle, le pilote bas niveau et la pile logicielle .

B. Mode périphérique et mode hôte

Dans une architecture USB, un système embarqué peut fonctionner en mode périphérique, en mode hôte, ou parfois en mode OTG selon les capacités matérielles. En mode périphérique, l'ARM Cortex se présente à l'ordinateur comme un équipement USB, par exemple un port série virtuel, un clavier, une mémoire de masse ou un périphérique personnalisé .

En mode hôte, c'est au contraire le microcontrôleur qui pilote le bus et gère un périphérique connecté, comme une clé USB ou un accessoire compatible. La documentation ST souligne justement la différence entre les piles Device et Host, et montre que leur initialisation implique des séquences et des responsabilités logicielles distinctes .

C. Structure d'une pile USB embarquée

Une pile USB embarquée s'organise généralement en plusieurs couches. La couche la plus basse initialise le contrôleur USB, les GPIO, les horloges, les interruptions et certains paramètres comme la vitesse, l'interface PHY ou la configuration des FIFO/endpoints . Au-dessus, la pile cœur implémente les mécanismes communs du protocole USB, puis les classes ajoutent le comportement spécifique d'un périphérique donné .

La documentation ST sur USB et USBX montre clairement cette séparation entre l'initialisation système de la pile et l'initialisation matérielle du périphérique USB. Certaines

fonctions initialisent la mémoire et les structures logicielles, mais l'utilisateur doit encore assurer la configuration bas niveau du contrôleur et des broches matérielles au niveau application/HAL .

D. Initialisation côté Device

Pour une application en mode périphérique sur STM32, l'initialisation de la pile USB Device commence par l'initialisation de la bibliothèque et l'enregistrement de la classe choisie, puis par la mise en place des couches basses assurant l'accès au contrôleur matériel . Le document ST précise aussi que certaines fonctions comme USBD_Init() supposent que le développeur implémente ou adapte les routines d'initialisation bas niveau, notamment pour les GPIO, l'horloge, les interruptions et le choix de l'interface matérielle .

Cette organisation illustre un point pédagogique essentiel : dans le développement USB, la réussite ne dépend pas uniquement de la pile logicielle fournie par le constructeur. Elle dépend aussi de la capacité du développeur à comprendre la carte, le schéma matériel, le contrôleur USB embarqué et les contraintes liées au mode Full Speed ou High Speed .

E. Initialisation côté Host

Le mode hôte USB demande un niveau de complexité supplémentaire, car le microcontrôleur doit détecter les connexions, alimenter éventuellement le bus, initialiser la communication et gérer les événements liés au périphérique branché. La documentation ST indique que USBH_Init() charge la bibliothèque hôte et s'appuie sur des couches HAL/LL pour configurer le matériel et les interruptions .

Dans cette configuration, l'application doit aussi prendre en charge certaines initialisations matérielles et callbacks d'événements. Cela montre qu'un développement USB hôte sur ARM Cortex exige non seulement la connaissance de la pile, mais aussi une bonne maîtrise des interruptions, du transfert de données et des scénarios de branchement/débranchement .

F. Classes USB courantes

Dans la pratique, plusieurs classes USB sont fréquemment utilisées dans les systèmes ARM Cortex. La classe CDC permet d'émuler un port série virtuel, très utile pour le débogage, la configuration ou l'acquisition de données depuis un PC. La classe HID permet de réaliser un

clavier, une souris ou un périphérique de commande sans pilote spécifique sur de nombreux systèmes d'exploitation. La classe MSC permet quant à elle de se comporter comme une mémoire de masse .

Le choix de la classe dépend du besoin applicatif. Pour un système de mesure, CDC est souvent suffisante ; pour un équipement de commande simple, HID peut être plus directe ; pour des échanges de fichiers, MSC est plus adaptée . Lorsque les besoins dépassent les classes standards, il est possible de recourir à une classe personnalisée, au prix d'une intégration logicielle plus complexe .

G. USB et RTOS

L'USB s'intègre très bien avec un RTOS comme FreeRTOS dans les applications complexes. Une tâche peut gérer la logique de haut niveau des échanges, tandis que les interruptions USB traitent les événements rapides puis notifient les tâches applicatives par queue ou sémaphore . Cette séparation améliore la lisibilité du code et réduit le travail fait directement dans les ISR, ce qui constitue une bonne pratique générale en temps réel. Dans un appareil embarqué avancé, il devient alors possible de faire coexister une acquisition temps réel, une connectivité IoT et une interface USB de configuration ou de maintenance .

H. Exemple d'application intégrée

Un exemple typique est un enregistreur de données basé sur ARM Cortex-M. FreeRTOS répartit le logiciel en tâches d'acquisition, de stockage, de communication et d'interface utilisateur ; la partie IoT transmet périodiquement des mesures ; et l'USB permet soit la configuration locale, soit l'export direct des fichiers de mesure vers un ordinateur .

Dans une telle architecture, l'USB ne remplace pas l'IoT, mais le complète. Il fournit une voie de maintenance locale rapide, souvent plus simple pour l'atelier, le laboratoire ou le diagnostic, tandis que la connectivité IoT sert à la supervision distante .

I. Articulation entre FreeRTOS, IoT et USB

Dans les systèmes embarqués modernes, ces trois domaines ne doivent pas être étudiés isolément. FreeRTOS apporte la structure d'exécution et le déterminisme logiciel ; l'IoT ouvre la

connectivité et la supervision distante ; l'USB fournit une interface locale normalisée pour la configuration, le diagnostic ou le transfert de données .

Sur un processeur ARM Cortex bien dimensionné, cette combinaison permet de concevoir des plateformes embarquées très complètes malgré des ressources limitées. L'ingénieur doit alors raisonner de manière globale : architecture matérielle, contraintes temps réel, consommation, sécurité, pile de communication et maintenabilité logicielle .

3.6 Points pédagogiques à retenir

- FreeRTOS transforme une application ARM Cortex complexe en ensemble de tâches concurrentes hiérarchisées par priorités, avec des mécanismes de synchronisation et de temporisation adaptés au temps réel .
- Le développement IoT sur ARM Cortex combine acquisition locale, traitement embarqué, communication réseau, gestion énergétique et robustesse logicielle .
- Le développement USB nécessite de maîtriser à la fois la pile protocolaire, l'initialisation matérielle et le choix des classes de périphériques ou du mode hôte/périphérique .
- Dans une architecture avancée, ces trois briques sont souvent combinées dans un même produit embarqué .

3.7 Conclusion

Les processeurs ARM-Cortex constituent une base particulièrement adaptée aux applications embarquées avancées grâce à la richesse de leur écosystème logiciel et matériel . L'utilisation de FreeRTOS améliore l'organisation et le comportement temps réel, le développement IoT étend les capacités du système vers la connectivité intelligente, et l'USB fournit une interface universelle de communication locale .

La maîtrise conjointe de ces trois dimensions permet de concevoir des systèmes embarqués modernes, robustes et évolutifs, capables de répondre à des besoins industriels, académiques ou de recherche. Pour un étudiant ou un ingénieur, comprendre leurs interactions est indispensable afin de passer d'une simple programmation de microcontrôleur à une véritable architecture embarquée professionnelle .